Parte OITO

Concorrência

Capítulo VINTE E SEIS

Conceitos Básicos de Threads

Objetivos do Exame

- Criar threads de trabalho usando Runnable, Callable e usar um ExecutorService para executar tarefas concorrentemente.
- Identificar possíveis problemas de threading entre deadlock, starvation, livelock e condições de corrida.

Threads

Em palavras simples, concorrência significa fazer coisas simultaneamente, em paralelo. Em Java, a concorrência é feita com threads.

Threads são unidades de código que podem ser executadas ao mesmo tempo. Elas às vezes são chamadas de processos leves, embora, na verdade, uma thread seja executada dentro de um processo (e todo processo tem, pelo menos, uma thread, a thread principal).

Em um nível baixo, podemos criar uma thread de duas maneiras.

A primeira (e recomendável) maneira é implementar a interface java.lang.Runnable, que possui apenas o método run():

```java
class RunnableTask implements Runnable {
    public void run() {
        System.out.println("Running");
    }
}
```

E então, para criar uma única thread, você precisa passar uma instância para o construtor da classe java.lang.Thread e SOLICITAR que a thread inicie (ela pode não iniciar imediatamente. Além disso, a ordem e o tempo de execução de uma thread NÃO são garantidos):

```java
Thread thread = new Thread(new RunnableTask());
```

```java
thread.start();
```

O método start() chamará o método run() da instância Runnable para começar a execução.

Claro, você pode usar uma classe anônima para fazer isso:

```java
Thread thread = new Thread(new RunnableTask() {
    public void run() {
        System.out.println("Running");
    }
});
thread.start();
```

Ou, já que Runnable é uma interface funcional, uma expressão lambda:

```java
Thread thread = new Thread(() -> {
    System.out.println("Running");
});
thread.start();
```

A outra maneira (desencorajada) é estender Thread, que implementa Runnable, então você apenas precisa sobrescrever run():

```java
class ThreadTask extends Thread {
    public void run() {
        System.out.println("Running");
    }
}
```

Então crie uma instância e chame o método start():

```java
Thread thread = new ThreadTask();
thread.start();
```

No entanto, é melhor implementar Runnable por conta própria porque, com a nova API de concorrência, você não precisa mais criar objetos Thread diretamente (sem mencionar que implementar uma interface é a maneira recomendada orientada a objetos de fazer isso).

ExecutorService

O Java 5 introduziu uma API de alto nível para concorrência, a maior parte dela implementada no pacote java.util.concurrent.

Uma das funcionalidades desta API são as interfaces Executor, que fornecem uma maneira alternativa (melhor) de iniciar e gerenciar threads.

O pacote java.util.concurrent define três interfaces de executor:

- **Executor**
  Esta interface possui o método execute(), que foi projetado para substituir:

```java
Runnable r = ...
Thread t = new Thread(r);
t.start();
```

por:

```java
Runnable r = ...
Executor e = ...
e.execute(r);
```

- **ExecutorService**
  Esta interface estende Executor para fornecer mais funcionalidades, como o método submit(), que aceita objetos Runnable e Callable e permite que retornem um valor.
- **ScheduledExecutorService**
  Esta interface estende ExecutorService para executar tarefas em intervalos repetidos ou com um atraso específico.

Executors usam *thread pools* (pools de threads), que usam *worker threads* (threads de trabalho). Essas threads são diferentes das threads que você cria com a classe Thread.

Quando *worker threads* são criadas, elas apenas ficam ociosas, esperando por trabalho. Quando o trabalho chega, o executor o atribui às threads ociosas do *thread pool*.

Dessa forma, as threads são genéricas, elas existem independentemente das tarefas Runnable que executam (em contraste com uma thread tradicional criada com a classe Thread).

Um tipo de *thread pool* é o *fixed thread pool*, que possui um número fixo de threads em execução. Se uma thread for encerrada enquanto ainda está em uso, ela é automaticamente substituída por uma nova thread. Também existem *thread pools* expansíveis.

Na maioria das vezes, você vai querer trabalhar com ExecutorService, já que ele tem mais funcionalidades que Executor. Como são interfaces, para criar uma instância de um Executor, você precisa usar uma classe auxiliar: java.util.concurrent.Executors.

A classe Executors possui muitos métodos estáticos para criar um ExecutorService, como:

```java
static ExecutorService newSingleThreadExecutor()
```

Cria um executor que usa uma única *worker thread*,

```java
static ExecutorService newFixedThreadPool(int nThreads)
```

Cria um *thread pool* que reutiliza um número fixo de threads, e

```java
static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)
```

Cria um *thread pool* que pode agendar tarefas.

Vamos começar criando um ExecutorService com um único *thread pool*:

```java
ExecutorService executor = Executors.newSingleThreadExecutor();
Runnable r = () -> {
    IntStream.rangeClosed(1, 4)
        .forEach(System.out::println);
};
System.out.println("before executing");
executor.execute(r);
System.out.println("after executing");
executor.shutdown();
```

Uma possível saída pode ser:

```pgsql
before executing
after executing
1
2
3
4
```

Como há apenas uma thread (além da thread principal, não se esqueça disso), as tarefas são garantidas de serem executadas na ordem em que foram submetidas, e no máximo uma tarefa estará ativa em qualquer momento. Em uma aplicação do mundo real, você pode querer usar newFixedThreadPool() com tamanho de pool igual ao número de processadores disponíveis.

Uma vez que você termina com um ExecutorService, você precisa encerrá-lo para finalizar as threads e liberar os recursos. Temos dois métodos para isso:

```java
void shutdown()
List<Runnable> shutdownNow()
```

O método shutdown() informa ao executor para parar de aceitar novas tarefas, mas as tarefas anteriores são permitidas a continuar até o fim. Durante esse tempo, o método isTerminated() retornará false até que todas as tarefas sejam concluídas, enquanto o método isShutdown() retornará true o tempo todo.

O método shutdownNow() também informará ao executor para parar de aceitar novas tarefas, mas TENTARÁ interromper todas as tarefas em execução imediatamente (interrompendo as threads, mas se a thread não responder à interrupção, ela pode nunca ser encerrada) e retornará uma lista das tarefas que nunca foram iniciadas.

Por exemplo, se executarmos:

```java
ExecutorService executor = Executors.newSingleThreadExecutor();
Runnable r = () -> {
    try {
        Thread.sleep(5_000);
    } catch (InterruptedException e) {
        System.out.println("Interrupted");
    }
};
executor.execute(r);
executor.shutdown();
```

Teremos que esperar cerca de cinco segundos para o programa terminar, mas se mudarmos shutdown() para shutdownNow(), "Interrupted" será impresso e o programa será encerrado imediatamente.

Além de execute(), existem outros métodos para submeter uma tarefa. Vamos definir, primeiro, os métodos mais críticos e as novas classes que eles usam:

```java
Future<?> submit(Runnable task)
```

Executa um Runnable e retorna um Future representando essa tarefa.

```java
<T> Future<T> submit(Callable<T> task)
```

Executa um Callable e retorna um Future representando o resultado da tarefa.

```java
<T> T invokeAny(
      Collection<? extends Callable<T>> tasks)
  throws InterruptedException, ExecutionException
```

Executa as tarefas fornecidas, retornando o resultado de uma que foi concluída sem lançar uma exceção, se houver. As outras tarefas são canceladas.

```java
<T> List<Future<T>> invokeAll(
      Collection<? extends Callable<T>> tasks)
  throws InterruptedException
```

Executa as tarefas fornecidas, retornando uma lista de objetos Future contendo seus status e resultados quando todas forem concluídas (normalmente ou por exceção).

A classe java.util.concurrent.Future possui estes métodos:

- boolean cancel(boolean mayInterruptIfRunning)
  Tenta cancelar a execução da tarefa. Se o argumento for true, a thread é interrompida. Caso contrário, a tarefa é permitida a concluir.
- V get() throws InterruptedException, ExecutionException
  Espera a tarefa concluir (indefinidamente), e então recupera seu resultado.
- V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException
  Espera, no máximo, o tempo fornecido e então recupera o resultado (se o tempo for alcançado e o resultado não estiver pronto, uma TimeoutException é lançada).
- boolean isCancelled()
  Retorna true se essa tarefa foi cancelada antes de completar normalmente.
- boolean isDone()
  Retorna true se a tarefa foi concluída.

java.util.concurrent.TimeUnit é um enum com os seguintes valores:

```java
TimeUnit.NANOSECONDS
TimeUnit.MICROSECONDS
TimeUnit.MILLISECONDS
TimeUnit.SECONDS
TimeUnit.MINUTES
TimeUnit.HOURS
TimeUnit.DAYS
```

java.util.concurrent.Callable é uma interface funcional que define este método:

```java
V call() throws Exception;
```

Como pode ver, a diferença com Runnable é que um Callable pode retornar um valor e lançar uma exceção verificada.

Agora os exemplos de código. Quando o método submit() é chamado com um Runnable, o objeto Future retornado devolve null (porque Runnable não retorna um resultado):

```java
ExecutorService executor = Executors.newSingleThreadExecutor();
Runnable r = () -> {
    IntStream.range(1, 1_000_000).forEach(System.out::println);
};
Future<?> future = executor.submit(r);
try {
    // Bloqueia até que o Runnable tenha terminado
    future.get();
} catch (InterruptedException | ExecutionException e) {
    /** ... */
}
```

Quando este método é chamado com um Callable, o objeto Future retornado contém o resultado quando tiver terminado a execução:

```java
ExecutorService executor = Executors.newSingleThreadExecutor();
Callable<Long> c = () ->
    LongStream.rangeClosed(1, 1_000_000).sum();
Future<Long> future = executor.submit(c);
try {
    // Bloqueia por 1 segundo até que o Callable tenha terminado
    Long result = future.get(1, TimeUnit.SECONDS);
} catch (InterruptedException | ExecutionException | TimeoutException e) {
    /** ... */
}
```

Assumindo a seguinte lista de objetos Callable:

```java
List<Callable<String>> callables = new ArrayList<>();
callables.add(() -> "Callable 1");
callables.add(() -> "Callable 2");
callables.add(() -> "Callable 3");
```

invokeAny() executa as tarefas fornecidas retornando o resultado de uma que tenha sido concluída com sucesso. Você não tem garantia sobre qual resultado do Callable você obterá — apenas um dos que terminarem:

```java
ExecutorService executor = Executors.newFixedThreadPool(3);
try {
    String result = executor.invokeAny(callables);
    System.out.println(result);
} catch(InterruptedException | ExecutionException e) {
    /** ... */
}
```

Às vezes isso imprimirá "Callable 1", às vezes "Callable 2", e outras vezes "Callable 3".

invokeAll() executa as tarefas fornecidas retornando uma lista de objetos Future que manterão o status e os resultados até que todas as tarefas estejam completas. Future.isDone() retorna true para cada elemento da lista retornada:

```java
ExecutorService executor = Executors.newFixedThreadPool(3);
try {
    List<Future<String>> futures = executor.invokeAll(callables);
    for(Future<String> f : futures){
        System.out.format("%s - %s%n", f.get(), f.isDone());
    }
} catch(InterruptedException | ExecutionException e) {
    /** ... */
}
```

Uma possível saída:

```nginx
Callable 1 - true
Callable 2 - true
Callable 3 - true
```

Problemas com Threads

Quando uma aplicação tem duas ou mais threads que a fazem se comportar de maneira inesperada, bem, isso é um problema.

Geralmente, a causa desse problema é que as threads são executadas em paralelo de tal maneira que, às vezes, elas competem para acessar recursos e, outras vezes, as ações de uma thread causam efeitos colaterais sobre as ações de outra (como modificar ou excluir alguns valores compartilhados).

Uma solução para esse problema é o conceito de *locking* (bloqueio), onde algo como um recurso ou um bloco de código é bloqueado com algum mecanismo de forma que apenas uma thread por vez possa usá-lo ou acessá-lo.

No entanto, se não tomarmos cuidado, o *locking* pode se tornar um dos seguintes três problemas:

Deadlock

Em palavras simples, uma situação de deadlock ocorre quando duas ou mais threads ficam bloqueadas para sempre, esperando que a outra libere ou adquira algum recurso.

Por exemplo, digamos que você e eu entramos em uma grande discussão — você diz que concorrência é difícil de fazer certo e eu digo que é fácil. Estamos realmente irritados um com o outro.

Depois de um tempo, você está pronto para dizer "Tudo bem, pode ser fácil" para acabar com a discussão, mas só se eu disser que estou errado. Da mesma forma, estou disposto a dizer que estou errado, que não é fácil, mas só se você disser que pode ser fácil.

Você está segurando o bloqueio sobre "Pode ser fácil" e está esperando que eu libere o bloqueio sobre "Não é fácil".

Eu estou segurando o bloqueio sobre "Não é fácil" e estou esperando que você libere o bloqueio sobre "Pode ser fácil".

Mas nenhum de nós vai admitir que está errado (liberar o bloqueio que temos), porque, quem faria isso? Então vamos ficar esperando um pelo outro (para sempre?). Isso é uma situação de *deadlock*.

Starvation (Inanição)

Starvation ocorre quando uma thread está constantemente esperando por um bloqueio, sem nunca conseguir obtê-lo porque outras threads com prioridade mais alta continuam adquirindo-o.

Suponha que você está no supermercado, esperando na fila do caixa. Então, um cliente com assinatura VIP chega e é atendido primeiro sem esperar. E então, chega outro cliente VIP. E depois outro. E você apenas espera, para sempre. Isso é *starvation*.

Livelock

Um *livelock* é como um *deadlock*, no sentido de que duas (ou mais) threads estão bloqueando uma à outra, mas, em um *livelock*, cada thread tenta resolver o problema por conta própria (*live*) em vez de apenas esperar (*dead*). Elas não estão bloqueadas, mas não conseguem fazer progresso.

Suponha que estamos andando em um beco estreito, em direções opostas. Quando nos encontramos, cada um de nós se move para o lado para deixar o outro passar, mas acabamos nos movendo para o mesmo lado ao mesmo tempo, repetidamente. Isso é um *livelock*.

Race Condition (Condição de Corrida)

Há outro problema de *threading* relacionado com como a concorrência funciona, e alguns o consideram a causa raiz dos problemas anteriores.

Uma *race condition* é uma situação em que duas threads competem para acessar ou modificar o mesmo recurso ao mesmo tempo, de uma forma que causa resultados inesperados (geralmente, dados inválidos).

Digamos que há um filme que você e eu queremos assistir. Cada um em sua própria casa, acessamos o site do cinema para comprar um ingresso. Quando verificamos a disponibilidade, só resta um. Ambos nos apressamos e clicamos no botão de compra ao mesmo tempo. Em teoria, podem acontecer três resultados:

- Ambos conseguimos comprar o ingresso
- Apenas um de nós consegue o ingresso
- Nenhum de nós consegue o ingresso (alguma outra pessoa o compra)

Isso é uma *race condition* (na maioria das *race conditions* há uma leitura e depois uma escrita).

A solução definitiva para esse problema é nunca modificar uma variável (por exemplo, tornando-as imutáveis). No entanto, isso nem sempre é possível.

Outra solução para evitar *race conditions* é realizar as operações de leitura e escrita de forma atômica (juntas em um único passo). Outra solução (também eficaz para os outros problemas) é garantir que a parte onde o problema acontece seja executada por apenas uma thread por vez de maneira apropriada.

No próximo capítulo, veremos como implementar essas soluções com a API de concorrência que o Java fornece.

Pontos-Chave

- Em um nível baixo, podemos criar uma thread de duas maneiras: implementando Runnable ou estendendo Thread e sobrescrevendo o método run().
- Em um nível alto, usamos Executors, que usam *thread pools*, que por sua vez usam *worker threads* (threads de trabalho).
- Um tipo de *thread pool* é o *fixed thread pool*, que possui um número fixo de threads em execução. Também podemos usar pools de thread únicos (*single-thread pools*).
- ExecutorService possui métodos para executar *thread pools* que recebem tarefas Runnable ou Callable. Uma Callable retorna um resultado e lança uma exceção verificada (*checked exception*).
- O método submit() retorna um objeto Future que representa o resultado da tarefa (se a tarefa for um Runnable, null é retornado).
- Um executor precisa ser encerrado para fechar o pool de threads com shutdown() (graciosamente) ou shutdownNow() (forçadamente).
- Uma situação de *deadlock* ocorre quando duas ou mais threads ficam bloqueadas para sempre, esperando que a outra adquira/libere algum recurso.
- *Starvation* (inanição) acontece quando uma thread está constantemente esperando por um bloqueio, sem nunca conseguir obtê-lo porque outras threads com prioridade mais alta continuam adquirindo-o.
- Um *livelock* é como um *deadlock* no sentido de que duas (ou mais) threads estão se bloqueando, mas, em um *livelock*, cada thread tenta resolver o problema por conta própria (*viva*) em vez de apenas esperar (*morta*).
- Uma *race condition* é uma situação onde duas threads competem para acessar ou modificar o mesmo recurso ao mesmo tempo de maneira que causa resultados inesperados.

Autoavaliação (Self Test)

1. Dado:

```java
ExecutorService service =
        Executors.newFixedThreadPool(2);
Future result = service.submit(() -> 1);
```

Assumindo que isso compila corretamente, qual é o tipo da expressão lambda?
A. Runnable
B. Callable
C. Supplier
D. Function

---

2. Quais das seguintes afirmações são verdadeiras?

A. Ao trabalhar com Runnable, você não pode usar um objeto Future.
B. Executor implementa AutoCloseable, então ele pode ser usado em um bloco *try-with-resources*.
C. Uma tarefa Callable pode ser cancelada.
D. Pools de threads contêm threads genéricas.

---

3. Dado:

```java
Future future = executor.submit(callable);
future.get(3, TimeUnit.MILLISECONDS);
```

O que o método get() faz?

A. Pode retornar um valor, após no máximo 3 milissegundos. Caso contrário, uma exceção é lançada.
B. Pode retornar um valor, após no máximo 3 milissegundos. Caso contrário, null é retornado.
C. Retorna um valor após exatamente 3 milissegundos.
D. Bloqueia o programa por 3 milissegundos sem retornar nada.

---

4. Dado:

```java
public class Question_26_1 {
    private static Object A = new Object();
    private static Object B = new Object();
    public static void main(String[] args) {
        new Thread(() -> {
            acquireLock(A);
            System.out.println("Just acquired A");
            acquireLock(B);
            System.out.println("Just acquired B");
            releaseLock(B);
            releaseLock(A);
        }).start();
        new Thread(() -> {
            acquireLock(B);
            System.out.println("Just acquired B");
            acquireLock(A);
            System.out.println("Just acquired A");
            releaseLock(A);
            releaseLock(B);
        }).start();
    }
    private static void acquireLock(Object o) {
        // Código para adquirir bloqueio no objeto o
    }
    private static void releaseLock(Object o) {
        // Código para liberar bloqueio no objeto o
    }
}
```

Qual problema de threading é mais provável de ocorrer nesse código?

A. Condição de Corrida (Race Condition)
B. Deadlock
C. Livelock
D. Nenhum problema